DATE: Monday, March 04, 2002 Printable Copy Create Case

Set Name side by side	Query	Hit Count	Set Name result set
DB=USPT; PLUR=YES; OP=ADJ			
L9	L7 and table$	1	L9
L8	L7 and (table with lookup$)	0	L8
L7	L5 and(instruction adj cod$)	1	L7
L6	L5 and (alternat$ with instruction with coding)	0	L6
L5	L3 and (interrupt$ with transfer$ with control$)	1	L5
L4	L3 and (cod$)	12	L4
L3	l1 and (interrupt$)	12	L3
L2	L1 and (alternate with coding)	0	L2
L1	(dual adj instruction adj set)	24	L1

END OF SEARCH HISTORY

Search

US005481684A

United States Patent [19]

Richter et al.

[11] Patent Number: 5,481,684

[45] Date of Patent: Jan. 2, 1996

[54] **EMULATING OPERATING SYSTEM CALLS IN AN ALTERNATE INSTRUCTION SET USING A MODIFIED CODE SEGMENT DESCRIPTOR**

[75] Inventors: **David E. Richter**, San Jose; **Jay C. Pattin**, Redwood City; **James S. Blomgren**, San Jose, all of Calif.

[73] Assignee: **Exponential Technology, Inc.**, San Jose, Calif.

[21] Appl. No.: **277,905**

[22] Filed: **Jul. 20, 1994**

Related U.S. Application Data

[63] Continuation-in-part of Ser. No. 179,926, Jan. 11, 1994.

[51] **Int. Cl.**[6] **G06F 9/30**

[52] **U.S. Cl.** **395/375**; 395/500; 364/DIG. 1

[58] **Field of Search** 395/375, 500, 395/800, 700

[56] **References Cited**

U.S. PATENT DOCUMENTS

3,764,988	10/1973	Omishi	395/375
4,077,058	2/1978	Appell et al.	395/650
4,633,417	12/1986	Wilburn et al.	364/550
4,763,242	8/1988	Lee et al.	395/500
4,780,819	10/1988	Kashiwagi	395/500
4,794,522	12/1988	Simpson	395/500
4,812,975	3/1989	Adachi et al.	395/500
4,821,187	4/1989	Ueda et al.	395/375
4,841,476	6/1989	Mitchell et al.	395/500
4,876,639	10/1989	Mensch, Jr.	395/500
4,928,237	5/1990	Bealkowski et al.	395/500
4,942,519	7/1990	Nakayama	395/775
4,943,913	7/1990	Clark	395/700
4,972,317	11/1990	Buonomo et al.	395/375
4,992,934	2/1991	Portanova et al.	395/375
5,053,951	10/1991	Nusinov et al.	395/425
5,077,657	12/1991	Cooper et al.	395/500
5,097,407	3/1992	Hino et al.	395/375
5,136,696	8/1992	Beckwith et al.	395/375
5,167,023	11/1992	de Nicolas et al.	395/375
5,210,832	3/1993	Maier et al.	395/375
5,226,164	7/1993	Nadas et al.	395/800
5,230,069	7/1993	Brelsford et al.	395/400
5,241,636	8/1993	Kohn	395/375
5,255,379	10/1993	Melo	395/400
5,269,007	12/1993	Hanawa et al.	395/375
5,287,465	2/1994	Kurosawa et al.	275/375
5,291,586	3/1994	Jen et al.	395/500

OTHER PUBLICATIONS

Combining both micro-code and Hardwired control in RISC by Bandyophyay and Zheng, Sep. 1987 Computer Architecture News pp. 11-15.

Combining RISC and CISC in PC systems By Garth, Nov. 1991 IEEE publication (?) pp. 10/1 to 10/5.

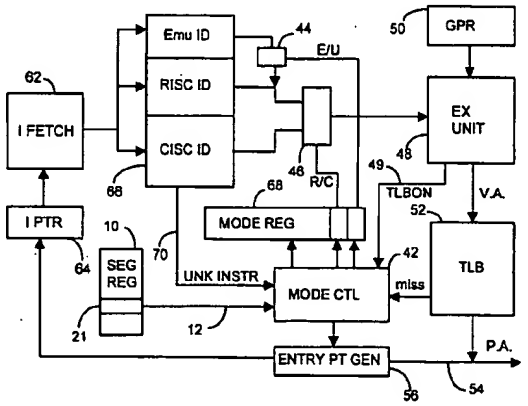
A 5.6-MIPS Call-Handling Processor for Switching Systems by Hayashi et al., IEEE JSSC Aug. 1989.

Primary Examiner—Parshotam S. Lall
Assistant Examiner—Viet Vu
Attorney, Agent, or Firm—Stuart T. Auvinen

[57] **ABSTRACT**

The CISC architecture is extended to provide for segments that can hold RISC code rather than just CISC code. These new RISC code segments have descriptors that are almost identical to the CISC segment descriptors, and therefore these RISC descriptors may reside in the CISC descriptor tables. The global descriptor table in particular may have CISC code segment descriptors for parts of the operating system that are written in x86 CISC code, while also having RISC code segment descriptors for other parts of the operating system that are written in RISC code. An undefined or reserved bit within the descriptor is used to indicate which instruction set the code in the segment is written in. An existing user program may be written in CISC code, but call a service routine in an operating system that is written in RISC code. Thus existing CISC programs may be executed on a processor that emulates a CISC operating system using RISC code. A processor capable of decoding both the CISC and RISC instruction sets is employed. The switch from CISC to RISC instruction decoding is triggered when control is transferred to a new segment, and the segment descriptor indicates that the code within the segment is written in the alternate instruction set.

6 Claims, 5 Drawing Sheets



WEST

Generate Collection

L9: Entry 1 of 1 File: USPT Jan 2, 1996

DOCUMENT-IDENTIFIER: US 5481684 A
TITLE: Emulating operating system calls in an alternate instruction set using a modified code segment descriptor

Abstract Paragraph Left (1):
The CISC architecture is extended to provide for segments that can hold RISC code rather than just CISC code. These new RISC code segments have descriptors that are almost identical to the CISC segment descriptors, and therefore these RISC descriptors may reside in the CISC descriptor tables. The global descriptor table in particular may have CISC code segment descriptors for parts of the operating system that are written in x86 CISC code, while also having RISC code segment descriptors for other parts of the operating system that are written in RISC code. An undefined or reserved bit within the descriptor is used to indicate which instruction set the code in the segment is written in. An existing user program may be written in CISC code, but call a service routine in an operating system that is written in RISC code. Thus existing CISC programs may be executed on a processor that emulates a CISC operating system using RISC code. A processor capable of decoding both the CISC and RISC instruction sets is employed. The switch from CISC to RISC instruction decoding is triggered when control is transferred to a new segment, and the segment descriptor indicates that the code within the segment is written in the alternate instruction set.

Parent Case Paragraph Right (1):
This application is a Continuation-in-Part of copending application for a "Dual-Instruction-Set Architecture CPU with Hidden Software Emulation Mode", filed Jan. 11, 1994, U.S. Ser. No. 08/179,926, having a common inventor and assigned to the same assignee as the present application.

Brief Summary Paragraph Right (1):
The present invention relates to a dual-instruction-set processor, and more particularly to a method and apparatus for emulating operating system calls using instructions from a second instruction set.

Brief Summary Paragraph Right (4):
A dual-instruction-set CPU was disclosed in the related application entitled "Dual-Instruction-Set Architecture CPU with Hidden Software Emulation Mode", filed Jan. 11, 1994, U.S. Ser. No. 08/179,926. That application is assigned to the same assignee as the present application. The dual-instruction set CPU contains hardware so that it can decode instructions from two entirely separate instruction sets.

Brief Summary Paragraph Right (6):
The present invention allows code from a first instruction set to

reside within a segment defined by a second instruction set. For example, RISC instruction code may reside within a CISC segment. The CISC architecture is extended to provide for segments that can hold RISC code or CISC code.

Brief Summary Paragraph Right (7):
In a broad sense the present invention is directed toward a segment descriptor for a dual-instruction-set processor. The processor executes instructions from a first instruction set and a second instruction set that are substantially independent. The segment descriptor describes a segment in memory containing program code. The segment descriptor has a location indicating means for indicating a location of the segment in the memory; attribute indicating means for indicating attributes of access to the segment; and an instruction set indicating means for indicating that an instruction set of the program code located within the segment belongs to one of a first instruction set and a second instruction set.

Brief Summary Paragraph Right (11):
The present invention allows an existing user program written in CISC code to call a service routine in an operating system that is written in RISC code. Thus existing CISC programs may be executed on a dual-instruction-set processor which can execute RISC code to emulate a CISC operating system.

Drawing Description Paragraph Right (1):
FIG. 1 shows the steps to service an x86 hardware interrupt.

Drawing Description Paragraph Right (5):
FIG. 5 is a block diagram of a dual-instruction-set CPU.

Detailed Description Paragraph Right (2):
A dual-instruction-set CPU was disclosed in the related application entitled "Dual-Instruction-Set Architecture CPU with Hidden Software Emulation Mode", filed Jan. 11, 1994, U.S. Ser. No. 08/179,926, hereby incorporated by reference. That application is assigned to the same assignee as the present application. The dual-instruction set CPU contains hardware so that it can decode instructions from two entirely separate instruction sets. A page fault or exception would cause the instruction set being decoded to switch. Thus if a page fault occurred when the CISC instruction set was being decoded, execution would switch to the RISC instruction set. CISC instructions that were not directly supported in hardware would also cause a switch to the RISC instruction set.

Detailed Description Paragraph Right (7):
The proper service routine is determined by accessing an interrupt table or interrupt descriptor table to fetch the starting address for the service routine for the particular exception. When an exception occurs, control is transferred to a service routine for the particular exception. The processor itself, however, supplies an entry number for an interrupt table.

Detailed Description Paragraph Right (8):
An external device may signal an interrupt to the processor. For example, a user may strike a key on the keyboard, which would generate a keyboard interrupt to the processor. The processor will perform an

external interrupt acknowledge cycle to allow the external device to identify an interrupt number. The interrupt number identifies an entry in the interrupt table which points to a service routine in the operating system for the external device.

Detailed Description Paragraph Right (9):
A wide variety of O/S support routines may be accessed by programming a software interrupt into the user code. A software interrupt is an instruction that emulates a hardware interrupt. The software interrupt instruction causes the interrupt table to be accessed. The software interrupt instruction has a parameter that specifies a unique entry in the interrupt table. When an interrupt is encountered, the interrupt table is consulted to determine the address where the interrupt service routine is located in memory. The processor loads this address and begins executing instructions from this address, the location of the service routine. Upon completion of the service routine, control is transferred or returned back to the user program. Application programs running under DOS typically use software interrupt instructions to invoke DOS routines.

Detailed Description Paragraph Right (10):
Application programs running under Windows.TM. occasionally use software interrupts to invoke operating system routines, but the bulk of the Windows.TM. operating system routines are invoked by a far call instruction. For example, a user application may call the "CreateWindow" command while running under Windows.TM. to have a new window opened. The user application program executes a far call instruction to transfer control to a different segment where the Windows.TM. CreateWindow routine is located. A far call is a transfer of control to code which resides in a different segment, which also saves the instruction pointer and code segment register onto a stack in memory. The CreateWindow routine returns to the application program by executing a far return instruction, which restores the instruction pointer and code segment from the stack.

Detailed Description Paragraph Right (11):
A segment descriptor is accessed and examined when a far call occurs, because control is transferred to a new segment. However, no interrupt is signaled.

Detailed Description Paragraph Right (12):
Many support routines supplied by the operating system are accessed when an external hardware interrupt is signaled to the processor. FIG. 1 shows the steps to service an x86 hardware interrupt. In the x86 architecture, only one pin or input to the processor is provided for most interrupts. Therefore, the processor must determine what the cause of the external interrupt is by generating an interrupt acknowledge cycle, when the external devices send an interrupt number or vector back to the processor. The interrupt vector specifies the device causing the interrupt, for example the keyboard. The interrupt vector is also known as an entry number, which specifies an entry in an interrupt table stored in memory. In the x86 architecture, the entry number is multiplied by eight, since each entry in the interrupt table occupies eight address locations, to specify the address of the entry in the interrupt table in memory. The entry stored in the interrupt table is a starting address where a support routine to service the interrupt is stored. The starting address of the interrupt

service routine is loaded into the processor's instruction pointer and code segment register, while the old values for the instruction pointer and code segment register, and the flags register, are stored on a stack in memory.

Detailed Description Paragraph Right (13):
The support routine is then executed starting with the instruction fetched from the starting address stored in the entry in the interrupt table. The support routine, or interrupt service routine, is executed, and control is returned to the user program when the end of the service routine is reached, by retrieving the old values for the instruction pointer, code segment and flags registers from the stack.

Detailed Description Paragraph Right (14):
As an example, the user may strike a key on the keyboard. The keyboard controller would signal to the processor an interrupt request over the shared interrupt input. The processor then "services" this interrupt. First, an interrupt acknowledge cycle is run when the keyboard's interrupt number, 09 hex, is supplied to the processor. The interrupt number is multiplied by 8 hex, and the result, 48 hex, is tadded to the interrupt descriptor table base register, yielding the address of the keyboard's entry in the interrupt table. A memory cycle is run at this address to fetch the contents of the interrupt descriptor table entry number at 48 hex, and the contents are stored in the processor. The old instruction pointer, code segment and flags registers are stored to the stack, and then the contents of the keyboard's entry from the interrupt table are loaded into the instruction pointer and code segment register. Execution then transfers to the keyboard interrupt service routine pointed to by the contents of the keyboard's entry from the interrupt table, which is a starting address for the keyboard service routine. This routine performs an I/O read of the keyboard to determine which key was struck, and then terminates and returns control to the user program by retrieving the old instruction pointer, code segment and flags registers from the stack.

Detailed Description Paragraph Right (15):
To service an x86 software interrupt, the steps are similar to those for the hardware interrupt of FIG. 1 except that an external interrupt acknowledge cycle is not necessary because the software interrupt instruction specifies the interrupt number and entry.

Detailed Description Paragraph Right (16):
The entries in the interrupt descriptor table are descriptors, similar to descriptors for segments. An offset address in the interrupt descriptor provides the entry point within the code segment jumped to. A selector field in the interrupt descriptor identifies the segment the interrupt service routine is located in. Privilege and access checks are performed for the interrupt descriptor just as they are done for segment descriptors. The interrupt descriptor table may contain a special descriptor, called a task gate, which causes the interrupt service routine to run in a separate context.

Detailed Description Paragraph Right (17):
A signal is needed to cause the processor to switch the instruction set being decoded. An exception or interrupt can provide this signal, or a separate instruction can be defined to switch instruction sets. Jumping from a CISC user program to another segment written in RISC

code without signaling an interrupt or exception could cause unpredictable results or even a system crash unless a method is employed to trigger the switch to RISC decoding. Additionally, routines within the operating system may jump to other operating system routines that may not be implemented in RISC code but in CISC code. Ideally the type of code, RISC or CISC, would be indicated when a jump or control transfer occurs, regardless of what caused the jump.

Detailed Description Paragraph Right (19):
Having two entry points for each O/S service routine is undesirable as it increases the memory requirement for the interrupt table. Performance would decrease because parameters or return values passed to and from the O/S service routine could have to be copied, saved, or re-arranged in registers or memory. One or more additional instructions would have to be executed, also reducing performance. Maintaining and verifying the operating system would be more difficult.

Detailed Description Paragraph Right (20):
Ideally either RISC or CISC code could use the same interrupt descriptor table and entry points. The O/S service routines would be independent of the user's instruction set.

Detailed Description Paragraph Right (21):
The inventors have recognized all of these operating system calls cause a control transfer to a different segment. The switch to the RISC instruction set is therefore best triggered by loading the new segment descriptor. Each segment is written in either RISC or CISC code, and its segment descriptor indicates the instruction set for the code in that segment. Thus if a jump occurs to a segment that has a descriptor indicating RISC code, then the processor will switch to RISC decoding if it is currently decoding CISC. The cause of the jump, be it an interrupt, exception, or merely a far jump to another segment, is irrelevant; the target segment type will cause the proper instruction set to be decoded for the new segment.

Detailed Description Paragraph Right (25):
Linear address 88 is translated to a physical address by translation-lookaside buffer or TLB 96, which is a small cache of the page translation tables stored in main memory. The TLB 96 translates the upper 20 bits of the linear address by searching the associative TLB cache for a match, and if one is found, then replacing these upper 20 bits with another 20 bits stored in the TLB 96.

Detailed Description Paragraph Right (26):
If the linear address is not found in the TLB, then a miss is signaled to the translator 98, which accesses the page tables in main memory and loads into the TLB the page table entry that corresponds to the linear address. Future references to the same page will "hit" in the TLB, which will provide the translation. Translator 98 may be implemented entirely in hardware, entirely in software, or in a combination of hardware and software.

Detailed Description Paragraph Right (29):
The system field 38 breaks segments into two broad classes: system segments that are used by the operating system, and user segments,

such as code, data, and stack segments. The Type field 36 further defines the type of segment pointed to by the descriptor. Some of the other attribute bits may change definition depending upon the segment type. Three bits are used to encode the type, so 2.sup.3 or 8 types are possible. For user segments, the type bits indicate if the segment is executable, writable, or readable. A code segment would be executable but not writable, while a data segment would be writable but not executable. For system segments, the accessed bit 40 is used as an extra type bit so that the type field is now 4 bits for system segments. The system segment types defined by Intel are shown in Table 1.

Detailed Description Paragraph Right (30):
FIG. 2 is a diagram of a gate descriptor in the x86 architecture. Gate descriptors control access to entry points into a code segment. Interrupt gate descriptors are placed in the interrupt descriptor table in protected mode. The gate descriptor 20 consists of two 4-byte double-words 20A and 20B. The beginning address of the service routine within the segment is determined by the offset 24, which is split among two fields, a first offset field 24A in the first double-word 20A, having bits 15 to 0 of the offset address, and a second offset field 24B in the second double-word 20B, having bits 31 to 16 of the offset address. Combining fields 24A and 24B yields a 32-bit offset address within the segment. A selector 22 identifies the segment that is the target of the gate descriptor. The target segment will have its own segment descriptor, such as the descriptor shown in FIG. 4, which must be accessed and checked before code can be fetched from the target segment.

Detailed Description Paragraph Right (32):
The type of gate can be interrupt, task switch, trap, or call, depending upon the type of control transfer defined by the gate. Table 1 also shows the types of gate descriptors defined for the x86 architecture. The last 4 rows of Table 1 are gate descriptor types while the first four rows of Table 1 are segment descriptor types.

Detailed Description Paragraph Right (33):
The x86 segment descriptors may be modified to indicate that the segment descriptor refers to a segment containing RISC code rather than x86 CISC code. An invalid or reserved combination of bits in the segment descriptor can be used to indicate that the processor should switch to decoding RISC code rather than CISC code when accessing code in this segment. Bit 21 in the second double-word of the segment descriptor of FIG. 4 is normally always zero for x86 systems. Setting this bit to one, which could cause a prior-art x86 system to perform an undocumented function, would indicate to a dual-instruction-set processor of the present invention that the segment contains code written in a RISC instruction set rather than the x86 CISC instruction set.

Detailed Description Paragraph Right (34):
Setting bit 21 to a one is the preferred technique for indicating RISC code within a segment because this bit can be set for any type of segment, system or user. However, other ways of indicating RISC code are also possible. Table 1A showed that four types of system segments were either invalid or reserved for Intel. Setting a descriptor for a system segment to one of these invalid or reserved types could also

indicate that the segment contains RISC code.

Detailed Description Paragraph Right (36):
Regardless of the reason for a control transfer, when a new segment is accessed the segment descriptor is checked to see if it indicates that the new segment contains RISC code or data. If so, then the processor will use a RISC instruction decoder rather than the CISC instruction decoder when executing instructions from the new segment. Any type of inter-segment transfer of control will force the processor to check the new segment descriptor to determine which instruction set to decode. Operating system calls from user code will cause an inter-segment jump, whether a software or hardware interrupt is used, or if a far jump directly to the address of the service routine is employed. The present invention will operate properly, checking the instruction set for the new segment, as long as the operating system is invoked by an inter-segment control transfer.

Detailed Description Paragraph Right (38):
The segment descriptors are stored in memory in tables. For the x86 architecture, a global descriptor table contains segment descriptors that are available to all tasks and users in a system. Each task or user will generally have its own local descriptor table storing segment descriptors for its own segments. Thus one user's segments are protected from another user because his segment descriptors are stored in his own local table. System descriptors are located in the global table, while user code, data, and stack segment descriptors are usually located in a user's local descriptor table. The interrupt table is usually shared by all users, and its entries are similar to segment descriptors. Rather than storing a segment base address and a limit, the interrupt descriptors contain an identifier to select a new segment, and an offset to specify a starting address to jump to within that segment.

Detailed Description Paragraph Right (39):
Using the present invention, RISC code can reside within a CISC segment. The CISC architecture is extended to provide for segments that can hold RISC code rather than just CISC code. These new RISC code segments have descriptors that are almost identical to the CISC segment descriptors, and therefore these RISC descriptors may reside in the CISC descriptor tables. The global descriptor table in particular may have CISC code segment descriptors for parts of the operating system that are written in x86 CISC code, while also having RISC code segment descriptors for the parts of the operating system that are written in RISC code.

Detailed Description Paragraph Right (40):
When control is passed to a new code segment, the segment descriptor is fetched from the global or local descriptor table, and protection checks are performed as usual. The present bit stored in the segment descriptor is examined, and an error is signalled if the segment is not present in memory. The type of the segment is checked, and an error is signalled if the segment is not a code segment. The privilege level in the descriptor is examined and a segment error is signaled if the privlege rules are violated. These protection checks are done without regard to the type of code residing in the segment, be it RISC or CISC.

Detailed Description Paragraph Right (42):
The next pages provide further background on the processor hardware used to implement a dual-instruction set processor. The present application is a Continuation-in-Part of the parent copending application for a "Dual-Instruction-Set Architecture CPU with Hidden Software Emulation Mode", filed Jan. 11, 1994, U.S. Ser. No. 08/179,926, having a common inventor and assigned to the same assignee as the present application.

Detailed Description Paragraph Right (44):
Instruction decode 66 is a partial instruction decode unit, in that it fully decodes only about 50% of the x86 CISC instructions, and about 85% of the PowerPC.TM. RISC instructions. Several well-known implementations are possible for instruction decode 66. For example, random logic may be used to decode the instruction set defined by an opcode map such as Tables 2 and 3. Opcode maps in Tables 2 and 3 are similar to logic truth tables in that they fully specify the logic equations needed to decode the instruction set. Instructions that are not fully decoded are not directly supported by hardware, and signal an "unknown opcode" on line 70 to mode control 42, which causes emulation mode to be entered.

Detailed Description Paragraph Right (49):
If the translation is not present in the TLB, a miss is signaled which causes emulation mode to be entered. Emulation mode is always used to load the TLB, allowing the emulation driver the highest level of control over address mapping and translation. Mode control 42 causes emulation mode to be entered whenever a miss is signaled from TLB 52, or an unknown opcode is detected by instruction decode 66. Normal exceptions, interrupts, and traps from the execute unit and other units also cause emulation mode to be entered, giving great flexibility in system design. Mode control 42 sets and clears the RISC/CISC and emulation mode control bits in mode register 68. When entry to emulation mode is requested, entry point block 56 generates the proper entry point vector or address in the emulation portion of memory, and loads this address into the instruction pointer 64. Thus the CPU will begin fetching and executing instructions at the specified entry point, where the emulation driver contains a routine to handle the exception, TLB miss, or to emulate the unknown instruction. Instruction decode 66 can provide the opcode itself and other fields of the instruction to the entry point logic, to allow the entry point to be more fully specified. Thus one entry point could be defined for a REP MOVS with a byte operand while another entry point is defined for a REP MOVS instruction with a long-word operand. Table 2 shows the entry points from CISC mode. For example, the REP MOVS byte instruction enters the emulation code at A4 hex, while REP MOVS longword enters at A5 hex. A TLB miss with segment 0 enters at 18 hex, while a far RETurn in x86 real mode enters at CA hex.

Detailed Description Paragraph Right (51):
The RISC sub-block of instruction decode 66 decodes the PowerPC.TM. RISC instruction set. All instructions are 32 bits in size, and some require two levels of instruction decoding. The first level determines the basic type of instruction and is encoded in the 6 most significant bits. Table 3 shows the 64 possible basic or primary opcode types. For example, 001110 binary (0E hex) is ADDI--add with an immediate operand, while 100100 (24 hex) is STW--store word. The CPU executes

the 45 unshaded opcodes directly in hardware. The fifteen darkly shaded opcodes, such as 000000, are currently undefined by the PowerPC.TM. architecture. Undefined opcodes force the CPU into emulation mode, where the emulation driver executes the appropriate error routine. Should instructions later be defined for these opcodes, an emulator routine to support the functionality of the instruction could be written and added to the emulator code. Thus the CPU may be upgraded to support future enhancements to the PowerPC.TM. instruction set. It is possible that the CPU could be field-upgradable by copying into emulation memory a diskette having the new emulation routine.

Detailed Description Paragraph Right (52):
The second level of instruction decoding is necessary for the remaining four lightly shaded opcodes of Table 3. Another 12-bit field in the instruction word provides the extended opcode. Thus one primary opcode could support up to 4096 extended opcodes. Primary opcode 010011, labeled "GRP A" in Table 3, contains instructions which operate on the condition code register, while groups C and D (opcodes 111011 and 111111 respectively) contain floating point operations. Group B (opcode 011111) contains an additional version of most of the primary opcode instructions, but without the displacement or immediate operand fields. Most group B and many instructions from groups A, C, and D are directly supported by the CPU's hardware, and the RISC instruction decoder thus supports some decoding of the 12-bit second level field. In the appendix is a list of the PowerPC.TM. instruction set, showing the primary and extended opcodes, and if the instruction is supported directly in hardware or emulated in emulation mode, as is, for example, opcode 2E, load multiple word.

Detailed Description Paragraph Right (53):
Extended instructions for controlling the CPU's hardware are added to the RISC instruction set by using undefined opcodes, such as those indicated by the darkly shaded boxes in Table 3. Thus additional logic may be added to the RISC instruction decode unit to support these additional instructions. However, user RISC programs must not be allowed to use these extended instructions. Therefore, the decoding of these extended instructions can be disabled for RISC user mode, and only enabled for emulation mode.

Detailed Description Paragraph Right (55):
CISC instructions can range in size from 8 bits (one byte) to 15 bytes. The primary x86 opcode, is decoded by the instruction decode 66 of FIG. 5. About 50% of the x86 instructions that can be executed by Intel's 80386 CPU are executed directly by the dual-instruction set CPU. Table 4 shows a primary opcode decode map for the x86 instruction set. Unshaded opcodes are directly supported in hardware, such as 03 hex, ADD r,v for a long operand. This same opcode, 03 hex, corresponds to a completely different instruction in the RISC instruction set. In CISC 03 hex is an addition operation, while in RISC 03 hex is TWI--trap word immediate, a control transfer instruction. Thus two separate decode blocks are necessary for the two separate instruction sets.

Detailed Description Paragraph Right (56):
A comparison of the opcode decoding of Table 3 for the RISC instruction set with Table 4 for the CISC instruction set shows that the two sets have independent encoding of operations to opcodes. While

both sets have ADD operations, the binary opcode number which encodes the ADD operation is different for the two instruction sets. In fact, the size and location of the opcode field in the instruction word is also different for the two instruction sets.

Detailed Description Paragraph Right (57):
Darkly shaded opcodes in Table 4 are not supported directly by hardware and cause an unknown or unsupported opcode to be signaled over line 70 of FIG. 5. This causes emulation mode to be entered, and the opcode is used to select the proper entry point in the emulation memory. By careful coding of the emulation routine, performance degradation can be kept to a minimum. Lightly shaded opcodes in Table 4 are normally supported directly by the CPU, but not when preceded by a repeat prefix (opcode F2 or F3).

Detailed Description Paragraph Left (3):
EXTERNAL INTERRUPTS

Detailed Description Paragraph Left (4):
SOFTWARE INTERRUPTS

Detailed Description Paragraph Left (6):
INTERRUPT SERVICE ROUTINES

Detailed Description Paragraph Left (7):
INTERRUPT TABLE DESCRIPTORS

Detailed Description Paragraph Type 1 (2):
External Interrupt

Detailed Description Paragraph Type 1 (3):
Software Interrupt

Detailed Description Paragraph Table (2):
TABLE 1 System Segment Types
Type Code Segment/gate 0
Invalid 1 286 TSS 2 LDT 3 286 TSS Busy 4 286 Call Gate 5 Task Gate 6 286 Interrupt Gate 7 286 Trap Gate 8 Invalid 9 486 TSS A Reserved by Intel B TSS Busy C 486 Call Gate D Reserved by Intel E 486 Interrupt Gate F 486 Trap Gate

Detailed Description Paragraph Table (3):
P Present bit. 1 = segment is valid; 0 = not valid DPL Descriptor Privilege Level 3-0 WD CNT Number of parameters passed to procedure called (call gate only) Type Segment Type (see Table 1)

Detailed Description Paragraph Table (4):
TABLE 3

PowerPC .TM. RISC Opcodes PowerPC primary opcode XXX000 XXX001 XXX010

000XXX ##STR62## ##STR63## ##STR64## TWI ##STR65## ##STR66## ##STR67## MULI 001XXX SUBFIC ##STR68## CMPLI CMPI ADDIC ADDIC. ADDI. ADDIS 010XXX BCx SC Bx ##STR69## RLWIMIX RLWINMx ##STR70## RLWNMx 011XXX ORI ORIS XORI XORIS ANDI. ANDIS. ##STR71## ##STR72## 100XXX LWZ LWZU LBZ LBZU STW STWU STB STBU 101XXX LHZ LHZU LHA LHAU STH STHU LMW STMW

110XXX LFS LFSU LFD LFDU STFS STFSU STFD STFDU 111XXX ##STR73## ##STR74## ##STR75## ##STR76## ##STR77## ##STR78## ##STR79## ##STR80##

Detailed Description Paragraph Table (6):
Hardware IU01 1F B mulhwux unsigned 1F 1CB divwx Divide word Hardware IU01 1F 1EB divwux unsigned ##STR267## 1F 215 lswx ##STR268## ##STR269## ##STR270## 1F 255 lswi ##STR271## ##STR272## ##STR273## 1F 295 stswx ##STR274## ##STR275## ##STR276## 1F 2D5 stswi ##STR277## ##STR278## ##STR279## Condition register instructions 13 0 mcrf Move CR field Hardware BU 13 21 crnor CR NOR Hardware BU 13 81 crandc CR AND with Hardware BU complement 13 C1 crxor CR XOR Hardware BU 13 E1 crnand CR NAND Hardware BU 13 101 crand CR AND Hardware BU 13 121 creqv CR Equivalent Hardware BU 13 1A1 crorc CR OR with complement Hardware BU 13 1C1 cror CR OR Hardware BU 1F 90 mtcrf Move to CR fields Hardware IU1 & BU 1F 200 mcrxr Move to CR from XER Hardware BU 3F 40 mcrfs Move to CR from Hardware BU FPSCR 1F 13 mfcr Move form CR field Hardware IU1 & BU 3F 26 mtfsb1x Move to FPSCR bit 1 Hardware BU 3F 46 mtfsb0x Move to FPSCR bit 0 Hardware BU 3F 86 mtfsfix Move to FPSCR Hardware BU immediate 3F 247 mffsx Move from FPSCR Hardware FU & BU 3F 2C7 mtfsfx Move to FPSCR Hardware FU & BU Privileged instructions 13 32 rfi Return from interrupt ##STR280## ##STR281## 13 96 isync Instruction synchronize ##STR282## ##STR283## ##STR284## ##STR285## ##STR286## ##STR287## ##STR288## ##STR289## ##STR290## ##STR291## ##STR292## ##STR293## ##STR294## ##STR295## 1F 53 mfmsr ##STR296## ##STR297## ##STR298## 1F 92 mtmsr ##STR299## ##STR300## ##STR301## 1F 132 tlbie TLB invalidate entry ##STR302## ##STR303## 1F 1F2 slbia SLB invalidate all ##STR304## ##STR305## 1F 1B2 slbia SLB invalidate entry ##STR306## ##STR307## 1F 1D2 slbiex SLB invalidate by index ##STR308## ##STR309## 1F 113 mftb Move from time base ##STR310## ##STR311## 1F 133 mftbu ##STR312## ##STR313## ##STR314## 1F 193 mttb Move to time base ##STR315## ##STR316## 1F 1B3 mttbu Move to time base upper ##STR317## ##STR318## 1F 153 mfspr ##STR319## ##STR320## ##STR321## 1F 1D3 mtspr ##STR322## ##STR323## ##STR324## ##STR325## ##STR326## ##STR327## ##STR328## Other user-mode instructions 1F 36 dcbst Data cache block store ##STR329## ##STR330## 1F 56 dcbf Data cache block flush ##STR331## ##STR332## 1F F6 dcbtst ##STR333## ##STR334## ##STR335## 1F 116 dcbt Data cache block touch ##STR336## ##STR337## 1F 1D6 dcbi ##STR338## ##STR339## ##STR340## 1F 3F6 dcbz Data cache block zero ##STR341## ##STR342## 1F 3D6 icbi ##STR343## ##STR344## ##STR345## 1F 356 eieio ##STR346## ##STR347## ##STR348## 1F 256 sync Synchronize ##STR349## ##STR350## 1F 136 eciwx ##STR351## ##STR352## ##STR353## 1F 1B6 ecowx ##STR354## ##STR355## ##STR356## Other instructions 1F 73 mfpmr Move from program mode register 1F B2 mtpmr Move to program mode register Floating point instructions 3B 12 fdivsx FP SP Divide Hardware FU 3B 14 fsubsx FP SP Subtract Hardware FU 3B 15 faddsx FP SP Add Hardware FU 3B 16 frsqrtsx FP SP Square root ##STR357## ##STR358## 3B 19 fmulsx FP SP Multiply Hardware FU 3B 1C fmsubsx FP SP Multiply-Subtract Hardware FU 3B 1D fmaddsx FP SP Multiply-Add Hardware FU 3B 1E fnmsubsx FP SP Neg-Mult-Subtract Hardware FU 3B 1F fnmaddsx FP SP Net-Mult-Add Hardware FU 3F 12 fdivx FP DP Divide Hardware FU 3F 14 fsubx FP DP Subtract Hardware FU 3F 15 faddx FP DP Add Hardware FU 3F 16 fsqrtx FP DP Square root ##STR359## ##STR360## 3F 19 fmulx FP DP Multiply Hardware FU 3F 1C fmsubx FP DP Multiply-Subtract Hardware FU 3F 1D fmaddx FP DP Multiply-Add Hardware FU 3F 1E fnmsubx FP DP Neg-Mult-Subtract Hardware FU 3F 1F fnmaddx

Detailed Description Paragraph Table Type 3 (1):
TABLE 2 CISC Entry Points x0 x1 x2 x3 x4 x5 x6 x7 x8 x9 xA xB xC xD xE xF 0x Undefined GRP7 LAR LSL CLTS ##STR1## ##STR2## ##STR3## ##STR4## ##STR5## ##STR6## ##STR7## ##STR8## ##STR9## ##STR10## 1x GRP6 ##STR11## ##STR12## ##STR13## ##STR14## ##STR15## ##STR16## ##STR17## ##STR18## ##STR19## ##STR20## 2x ##STR21## ##STR22## ##STR23## ##STR24## ##STR25## ##STR26## DAA ##STR27## ##STR28## ##STR29## ##STR30## ##STR31## ##STR32## ##STR33## DAS 3x AAA ##STR34## ##STR35## ##STR36## ##STR37## ##STR38## ##STR39## AAS 4x reset NMI INTR ##STR40## ##STR41## ##STR42## ##STR43## ##STR44## ##STR45## ##STR46## 5x ##STR47## ##STR48## ##STR49## ##STR50## ##STR51## ##STR52## ##STR53## 6x ##STR54## POPA ##STR55## ARPL INS INS OUTS OUTS 7x 8x SHLD SHLD ##STR56## ##STR57## RSM SHRD ##STR58## 9x LFS LGS LFS LGS Call Call PUSH POPF PUSH POP far far F F F real real prot prot real prot 16bit 16bit 32bit 32bit Ax PUSH POP Movs Movs Cmps Cmps PUSH POP Stos Stos Lods Lods Scas Scas FS FS GS GS byte long byte long byte long byte long byte long real REP REP REP REP real REP REP REP REP REP REP Bx POP LSS LSS Movs Movs Cmps Cmps POP Stos Stos Lods Lods Scas Scas FS GS byte long byte long byte long byte long byte long prot real prot repne repne repne repne prot repne repne repne repne repne repne Cx XAD XAD LES LDS LES LDS Enter Leave RET RET INT3 INTn INTO IRET D D far far real real prot prot real real real real real real Dx AAM AAD XLAT RET RET INT3 INTn INTO IRET far far prot prot prot prot prot prot Ex IN IN OUT OUT JMP JMP IN IN OUT OUT far far real prot Fx HLT ##STR59## ##STR60## CLI STI BSF BSR ##STR61##

Detailed Description Paragraph Table Type 3 (2):
TABLE 4 x86 CISC Opcode Map OP x0 x1 x2 x3 x4 x5 x6 x7 x8 x9 xA xB xC xD xE xF 0x ##STR81## ##STR82## ##STR83## ##STR84## ##STR85## ##STR86## ##STR87## ##STR88## ##STR89## ##STR90## ##STR91## ##STR92## ##STR93## ##STR94## ##STR95## ##STR96## 1x ##STR97## ##STR98## ##STR99## ##STR100## ##STR101## ##STR102## ##STR103## ##STR104## ##STR105## ##STR106## ##STR107## ##STR108## ##STR109## ##STR110## ##STR111## ##STR112## 2x ##STR113## ##STR114## ##STR115## ##STR116## ##STR117## ##STR118## ##STR119## ##STR120## ##STR121## ##STR122## ##STR123## ##STR124## ##STR125## ##STR126## ##STR127## ##STR128## 3x ##STR129## ##STR130## ##STR131## ##STR132## ##STR133## ##STR134## ##STR135## ##STR136## ##STR137## ##STR138## ##STR139## ##STR140## ##STR141## ##STR142## ##STR143## ##STR144## 4x INC INC INC INC INC INC INC INC DEC DEC DEC DEC DEC DEC DEC DEC AX CX DX BX SP BP SI DI AX CX DX BX SP BP SI DI 5x PUSH PUSH PUSH PUSH PUSH PUSH PUSH PUSH POP POP POP POP POP POP POP POP AX CX DX BX SP BP SI DI AX CX DX BX SP BP SI DI 6x ##STR145## ##STR146## ##STR147## ##STR148## ##STR149## ##STR150## ##STR151## ##STR152## ##STR153## ##STR154## ##STR155## ##STR156## ##STR157## ##STR158## ##STR159## ##STR160## 7x JO JNO JB JNB JZ JNZ JBE JNBE JS JNS JP JNP JL JNL JLE JNLE 8x ##STR161## ##STR162## ##STR163## ##STR164## ##STR165## ##STR166## ##STR167## ##STR168## ##STR169## ##STR170## ##STR171## ##STR172## ##STR173## LEA ##STR174## ##STR175## 9x NOP ##STR176## ##STR177## ##STR178## ##STR179## ##STR180## ##STR181## ##STR182## ##STR183## ##STR184## ##STR185## ##STR186## ##STR187## ##STR188## ##STR189## ##STR190## Ax ##STR191## ##STR192## ##STR193## ##STR194## ##STR195## ##STR196## ##STR197## ##STR198## ##STR199## ##STR200## ##STR201## ##STR202## ##STR203## ##STR204## ##STR205## ##STR206## Bx MOV MOV MOV MOV MOV MOV MOV MOV MOV MOV MOV MOV MOV MOV MOV MOV AL,i CL,i DL,i BL,i AH,i CH,i DH,i BH,i AX,i CX,i DX,i BX,i SP,i BP,i SI,i DI,i Cx ##STR207##

##STR208## ##STR209## RET ##STR210## ##STR211## ##STR212## ##STR213##
##STR214## ##STR215## ##STR216## ##STR217## ##STR218## ##STR219##
##STR220## ##STR221## Dx ##STR222## ##STR223## ##STR224## ##STR225##
##STR226## ##STR227## ##STR228## ##STR229## FP FP FP FP FP FP FP FP Ex
##STR230## ##STR231## Loop JCXZ ##STR232## ##STR233## ##STR234##
##STR235## ##STR236## ##STR237## ##STR238## ##STR239## ##STR240##
##STR241## ##STR242## ##STR243## Fx Lock ##STR244## REP ##STR245##
##STR246## ##STR247## ##STR248## ##STR249## ##STR250## ##STR251##
##STR252## ##STR253## ##STR254## ##STR255## ##STR256##

CLAIMS:

1. A method for emulating calls from a user program to an operating system, said method comprising:

executing a plurality of user instructions from said user program, said user instructions belonging to a first instruction set;

decoding a call instruction in said user program, said call instruction calling a service routine in an operating system, wherein said call instruction in said user program is a far jump instruction;

loading a pointer to a code segment, said code segment containing said service routine in said operating system, said pointer having an instruction set indicating means for indicating an instruction set for said service routine;

executing service routine instructions in said code segment, decoding service routine instructions with a first instruction decoder when said instruction set indicating means indicates said first instruction set, decoding service routine instructions with a second instruction decoder when said instruction set indicating means indicates a second instruction set, said first instruction decoder for decoding only a portion of said first instruction set;

returning control to said user program when a return instruction is executed in said service routine;

whereby said user program containing instructions in said first instruction set calls said service routine in said operating system, said service routine having instructions from said second instruction set, said pointer to said code segment indicating if said service routine contains instructions from said second instruction set or said first instruction set.

6. A method for emulating calls within a user program, said method comprising:

executing a plurality of user instructions from said user program, said user instructions belonging to a first instruction set;

decoding a call instruction in said user program, said call instruction calling a service routine in said user program, wherein said call instruction in said user program is a far jump instruction;

loading a pointer to a code segment, said code segment containing said service routine in said user program, said pointer having an

instruction set indicating means for indicating an instruction set for said service routine;

executing service routine instructions in said code segment, decoding service routine instructions with a first instruction decoder when said instruction set indicating means indicates said first instruction set, decoding service routine instructions with a second instruction decoder when said instruction set indicating means indicates a second instruction set, said first instruction decoder for decoding only a portion of said first instruction set;

returning control to said user program when a return instruction is executed in said service routine;

whereby said user program containing instructions in said first instruction set calls said service routine in said user program, said service routine having instructions from said second instruction set, said pointer to said code segment indicating if said service routine contains instructions from said second instruction set or said first instruction set.